Persistent `constexpr` allocation

Document #: P1974R1
Date: 2026-06-09
Project: Programming Language C++
Audience: EWG
Reply-to: Jeff Snyder
<jeff-isocpp@caffeinated.me.uk>
Daveed Vandevoorde
<daveed@edg.com>

Contents

1 Abstract **1**

2 Revision History **1**

3 Status of this paper **1**

4 Motivation **2**

5 Discussion **2**
5.1 Prior work 2
5.2 A mental model 2
5.3 Using data in `constexpr` allocations within constant expressions 3

6 Proposal **4**

7 Wording **5**

8 References **6**

1 Abstract

This paper proposes a set of constness-based requirements that govern when it is safe to allow persistent `constexpr` allocations, when the contents of such allocations are constant expressions, and when they can be placed in immutable storage.

2 Revision History

— R1: removed propconst from the paper

3 Status of this paper

This paper is a new core language feature proposal targeting C++29.

4 Motivation

Enabling persistent `constexpr` allocations in C++ unlocks the ability to construct complex data structures at compile time and store them in static storage for efficient access at runtime. This capability bridges a longstanding gap between compile-time computation and runtime efficiency, allowing developers to build data structures at compile time and use them at runtime without incurring runtime initialization costs. Such data structures can be used both for holding fixed data which can be accessed and used at both compile time and runtime, or they can be used to hold data which is read and/or mutated at runtime. The latter case allows developers to have structured mutable global storage in their programs, but comes with the restriction that the data cannot be used in constant expressions.

5 Discussion

5.1 Prior work

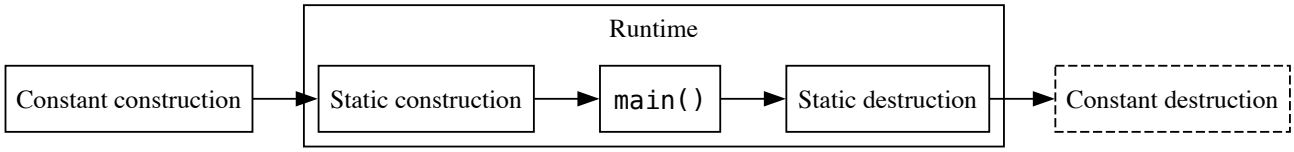
Promotion of persistent `constexpr` allocations to static storage was proposed by [P0784R5], but the feature was not accepted into C++20 due to concerns surrounding composability. For example, should the following snippet compile?

```
constexpr unique_ptr<unique_ptr<int>> uui
    = make_unique<unique_ptr<int>>(make_unique<int>());
int main() {
    unique_ptr<int>& ui = *uui;
    ui.reset();
}
```

As part of the test for whether a `constexpr` allocation can be promoted to static storage, [P0784R5] proposed that "evaluating that destructor would be a valid core constant expression and would deallocate all the persistent allocations produced by the evaluation of expr.", and the above example satisfies that part of the test. However, as P0784R5 also points out, the test is meaningless because the destructor reads from a variable that is mutable at runtime (the internal `int*` in the inner `unique_ptr<int>`), and so without further guarantees about its immutability being provided, it must be rejected. The paper goes on to propose an imperative method of guaranteeing that immutability.

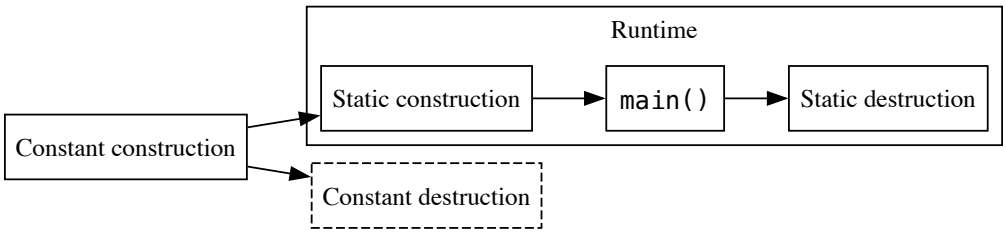
5.2 A mental model

Persistent `constexpr` allocations are made during the initialization of `constexpr` global variables, and their lifetimes are tied to the global variable whose initialization performed the allocation. We can think of program execution as having the following phases:



In terms of the execution of the compiled program, the constant construction and constant destruction phases are purely notional; there are no instructions in the compiled program corresponding to them. However, we can still think of them as being present for purposes of language specification.

To make sure there is no undefined behaviour during constant destruction, we need to evaluate the destructors of `constexpr` globals. However, there is a practical constraint in that the compiler only runs before program runtime, and not after. To work around this, we simulate the destruction of `constexpr` globals after their construction, thus our actual phases are:



This reordering is valid so long as all of the inputs to the constant destruction do not change during runtime. As with all `constexpr` evaluation, the destructors evaluated during constant destruction are only allowed to access other constant expressions, and so this reordering is valid. This is captured by the rules for persistent allocation from [P0784R5], which this paper uses largely unmodified:

> A `constexpr` variable initializer *expr* may create persistent `constexpr` allocations if:
>
> — The result of evaluating the initializer is an object with a nontrivial `constexpr` destructor, and
> — Evaluating that destructor would be a valid core constant expression and would deallocate all the persistent allocations produced by the evaluation of *expr*.

A notable side-effect of the requirement that all persistent allocations are deallocated during constant destruction is that any attempt to deallocate a persistent `constexpr` allocation during runtime would create a double deallocation during constant destruction, and is therefore ill-formed (no diagnostic required).

5.3 Using data in `constexpr` allocations within constant expressions

Due to the requirement that evaluation of the destructor is a valid core constant expression, the composability of persistent `constexpr` allocations hinges on how we determine whether the data within a `constexpr` allocation is constant or not.

Consider the following examples based on a much-simplified `unique_ptr`:

```
template <typename T>
struct ptr {
    T* value;
    constexpr ptr(T* value) : value{value} {}
    ptr(ptr&&) = delete;
    constexpr ~ptr() { delete value; }
};

constexpr ptr<int> pi = new int{1};  // OK
char str1[*pi.value]{};              // ill-formed: *pi.value is not a constant expression
int& pi_value = *pi.value;           // OK
```

`*pi.value` cannot be a constant expression, as it has type `int&` and is mutable at runtime. The only reasonable way to compile this is to make its storage part of the `.data` section (or the non-ELF equivalent thereof) in the resulting program.

```
constexpr ptr<const int> pci = new int{42};  // OK
char str2[*pci.value]{};                     // OK: *pci.value is a constant expression
int& pci_value = *pci.value;                 // ill-formed: discards qualifiers
```

Conversely, `*pci.value` has type `const int&`, and moreover there is no alternative way to access the same allocation as a non-`const` lvalue. Therefore, we can make `*pci.value` a constant expression and emit its storage as part of `.rodata` section instead of `.data`.

This leads to a rule for when variables in persistent `constexpr` allocations are usable in constant expressions:

If a persistent `constexpr` allocation is not *reachable as mutable* after evaluation of the `constexpr` variable initializer which made the allocation, variables within that allocation are immutable and usable in constant expressions.

When combined with the previous rule regarding creation of persistent `constexpr` allocations, this is sufficient to support simple use cases such as `constexpr vector<int>` and `constexpr string`, but it is insufficient for most nested allocations such as `constexpr vector<string>` and `constexpr vector<vector<int>>`.

The following examples illustrate this using nested instantiations of `ptr`. Both `constexpr vector<T>` and `constexpr basic_string<T>` are equivalent to `ptr<T>` for the purpose of understanding their interaction with persistent `constexpr` allocations, since they contain non-`const` pointers to their data.

```
constexpr ptr<ptr<int>> ppi            // ill-formed: in ~ptr<ptr<int>>, ppi.value->value
    = new ptr<int>{new int{1}};        //             is not a constant expression
constexpr ptr<ptr<const int>> ppci     // ill-formed: in ~ptr<ptr<const int>>, ppci.value->
    = new ptr<const int>{new int{42}}; //             value is not a constant expression
```

In these examples, `ppi.value` and `ppci.value` are both const pointers, but their pointees are not const. Therefore, `ppi.value->value` and `ppci.value->value` are reachable as mutable and not constant expressions. The destructors `~ptr<int>` and `~ptr<const int>` attempt to read these pointers (but not the `int` pointees) despite them not being constant expressions, and due to this they are both ill-formed.

```
constexpr ptr<const ptr<int>> pcpi    // OK: in ~ptr<const ptr<int>>, pcpi.value->value
    = new const ptr<int>{new int{1}}; //     is a constant expression
char str3[*pcpi.value->value]{};      // ill-formed: *pcpi.value->value is not a constant
                                      //             expression.
int& pcpi_value = *pcpi.value->value; // OK
```

Converting `ptr<int>` to `const ptr<int>` in `ppi`'s declaration makes it well-formed again, but the `int` pointee is still mutable and not a constant expression.

```
constexpr ptr<const ptr<const int>> pcpci     // OK: in ~ptr<const ptr<const int>>,
    = new const ptr<const int>{new int{42}};  //     pcpci.value->value is a constant
                                              //     expression
char str4[*pcpci.value->value]{};             // OK: *pcpci.value->value is a constant
                                              //     expression
int& pcpci_value = *pcpci.value->value;       // ill-formed: discards qualifiers
```

Converting `ptr<const int>` to `const ptr<const int>` in `ppci`'s declaration makes it well-formed again, and as with `pci`, the `int` is usable in constant expressions.

6 Proposal

This paper proposes that:

1. A `constexpr` variable initializer *expr* may create persistent `constexpr` allocations if:
   - The result of evaluating the initializer is an object with a nontrivial `constexpr` destructor, and
   - Evaluating that destructor would be a valid core constant expression and would deallocate all the persistent allocations produced by the evaluation of *expr*.
2. Variables in persistent `constexpr` allocations which are not reachable as mutable after evaluation of the `constexpr` variable initializer whose evaluation performed the allocation are usable in constant expressions. Variables in other persistent `constexpr` allocations are not usable in constant expressions and are mutable at runtime.

These two rules establish a foundation for persistent `constexpr` allocations, and allow for some simple use-cases including `constexpr string` and `vector`, with the limitation that their contents are not constant expressions.

To bridge the gap between this foundation and having `constexpr` strings and vectors whose contents are usable in constant expressions, as well as use cases involving nested `constexpr` allocations such as `vector<string>`, we need some approach that allows the compiler to know that our deep-const types are actually deep-const.

Many such options have been discussed, including:

- A magic function that blesses allocations as immutable (`mark_immutable_if_constexpr`) [P0784R5]
- A cv-qualifier for propagation of `const`ness [P1974R0]
- A decl-specifier for propagation of `const`ness [P2670R1]
- Blessing `vector` and `string` as deep-const [P3554R0]
- Allowing the compiler to infer deep constness via inspection of the code that accesses the `constexpr` allocation (informal discussion only)

This paper does not propose any specific solution in this space, leaving it as a problem for future proposals.

7 Wording

Change paragraph 2 of §7.7.2 [expr.const.core] in [N5046]:

An expression `E` is a *core constant expression* unless the evaluation of `E`, following the rules of the abstract machine, would evaluate one of the following:

- …
- an lvalue-to-rvalue conversion that is applied to a glvalue that refers to a non-active member of a union or a subobject thereof;
- an lvalue-to-rvalue conversion that is applied to a glvalue designating an object in a persistent `constexpr` allocation which is reachable as mutable from the `constexpr` variable whose initialization created that allocation;
- an lvalue-to-rvalue conversion that is applied to an object with an indeterminate value;
- …
- a new-expression, unless either
  - the selected allocation function is a replaceable global allocation function and the allocated storage is deallocated either within the evaluation of E or by the destruction of the variable with static storage duration whose initializer allocated it, or
  - the selected allocation function is a non-allocating form (17.6.3.4) with an allocated type T, where
    - the placement argument to the new-expression points to an object that is pointer-interconvertible with an object of type T or, if T is an array type, with the first element of an object of type T, and
    - the placement argument points to storage whose duration began within the evaluation of E;
- …

Add after paragraph 3 of §7.7.7 [expr.const.defns] in [N5046]:

4 A *persistent constexpr allocation* is a region of storage A obtained by an invocation of an allocation function during the initialization of a `constexpr` variable V of class type or (possibly multi-dimensional) array thereof, but not passed to a deallocation function during that initialization, such that the hypothetical expression considered when determining whether V has constant destruction ([expr.const.defns]) would deallocate A. Such a persistent `constexpr` allocation is immutable unless it is reachable as mutable from V or there exists an object stored in A (or subobject thereof) which is declared mutable. If A is deallocated, except by the implicit invocation of V's destructor, the behavior is undefined.

5 A persistent `constexpr` allocation A created during the initialization of a `constexpr` variable V is *reachable as mutable* from V if V, a subobject of V, an object stored in any of V's persistent `constexpr` allocations, or any subobject thereof, is a pointer or reference to either A, an object stored in A, or any subobject thereof, and the pointer or reference is to either a non-`const`-qualified type or a type containing `mutable` members.

8 References

[N5046] Thomas Köppe. 2026-05-12. Working Draft, Programming Languages — C++.
https://wg21.link/n5046

[P0784R5] Peter Dimov, Louis Dionne, Nina Ranns, Richard Smith, Daveed Vandevoorde. 2019-01-21. More constexpr containers.
https://wg21.link/p0784r5

[P1974R0] Jeff Snyder, Louis Dionne, Daveed Vandevoorde. 2020-05-15. Non-transient constexpr allocation using propconst.
https://wg21.link/p1974r0

[P2670R1] Barry Revzin. 2023-02-03. Non-transient constexpr allocation.
https://wg21.link/p2670r1

[P3554R0] Barry Revzin, Peter Dimov. 2025-01-06. Non-transient allocation with vector and basic_string.
https://wg21.link/p3554r0